Unifying Agent Systems

Mark d'Inverno
Cavendish School of Computer Science
University of Westminster
London, W1M 8JS, UK
dinverm@wmin.ac.uk

Michael Luck
Dept of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, UK
mml@ecs.soton.ac.uk

Abstract

Whilst there has been an explosion of interest in multi-agent systems, there are still many problems that may have a potentially deleterious impact on the progress of the area. These problems have arisen primarily through the lack of a common structure and language for understanding multi-agent systems, and with which to organise and pursue research in this area. In response to this, previous work has been concerned with developing a computational formal framework for agency and autonomy which, we argue, provides an environment in which to develop, evaluate, and compare systems and theories of multi-agent systems. In this paper we go some way towards justifying these claims by reviewing the framework and showing what we can achieve within it by developing models of agent dimensions, categorising key inter-agent relationships and by applying it to evaluate existing multi-agent systems in a coherent computational model. We outline the benefits of specifying each of the systems within the framework and consider how it allows us to unify different systems and approaches in general.

1 Introduction

In recent years, there has been an explosion of interest in agents and multi-agent systems. This has not only been in artificial intelligence but in other areas of computer science such as information retrieval and software engineering. Indeed, there is now a plethora of different labels for agents including *autonomous agents* [32], *software agents* [22], *intelligent agents* [59], *interface agents* [37], *virtual agents* [1], *information agents* [35], *mobile agents* [57], and so on. The diverse range of applications for which agents are being touted include operating systems interfaces [21], processing satellite imaging data [54], electricity distribution management [31], air-traffic control [34] business process management [29], electronic commerce [26] and computer games [25], to name a few. Moreover, significant commercial and industrial research and development efforts have been underway for some time [9, 11, 44, 45], and are set to grow further.

However, the field of agents and multi-agent systems is still relatively young, and there are many problems that may have a potentially deleterious impact on the progress of the area. These problems have arisen primarily through the lack of a common structure and language for understanding multi-agent systems, and with which to organise and pursue research in multi-agent systems. It is, therefore, important to ensure that any such structures we generate are *accessible* if there are going to have any significant impact on the way research progresses [38]. In particular, we need to be able to relate different theories and approaches within MAS so that different systems and models can be integrated. This can be achieved in two stages: first we need to be able to isolate the potential inconsistencies in

definitions of fundamental terms frequently used when discussing multi-agent systems, and second we need to provide an environment in which different systems and theories can be developed, evaluated and compared.

1.1 Formal Frameworks

We have previously considered the requirements for the structures or *frameworks* that are necessary to provide a rigorous approach to any discipline [19], and in particular to agents and multi-agent systems [40]. Such frameworks should precisely and unambiguously provide meanings for common concepts and terms but in an *accessible* manner since only then will a common conceptual framework have a chance of emerging. (If there is a generally held understanding of the salient features and issues involved in the relevant class of models then we can assert the existence of a common conceptual framework). Another important aspect, and key to the work presented in this paper is that it enables models and systems to be explicitly presented, compared and evaluated. Not only must it provide a description of the common abstractions found within that class of models, but also it must provide a means of further refinement to specify particular models and systems.

Multi-agent systems are inherently complicated, and, consequently, reasoning about the behaviour of such systems becomes difficult. The ability to formalise multi-agent systems, to do so in such a way that allows automated reasoning about agents' behaviour and, additionally, to do so in a way that is also *accessible*, is therefore critically important.

In what follows, we seek to show that the formal framework we have developed satisfies these requirements and provides just such a base as is necessary for a rigorous and disciplined approach to multi-agent systems research. Our aim here is not to provide a detailed presentation of our framework, which has been presented extensively elsewhere, but instead to show how it may be applied to different systems, and how they may be accommodated within a single overarching framework. Similarly, we are not concerned in this paper with the detailed specification of agent behaviour, though we have addressed this previously in [16, 18], nor with reasoning about agent behaviour, though again related work has addressed this, for example in the context of Agentis [15, 56].

1.2 Overview

In this paper, we review and build on previous work that has developed a formal agent framework and extend it to construct several agent models. These models range from generic abstract models to very specific system models that have been described and implemented elsewhere. In the next section we review the framework we have developed and in Section 4 and show how it can be extended to describe certain types of agents, describing autonomous agents, planning agents, memory agents and social agents. Next, the paper presents three case studies in applying the framework. These case studies have been chosen as exemplars specifically because they lie at opposite ends of the multi-agent system spectrum; one is a relatively new mostly theoretical model, while the other two are well-known, fully developed and implemented systems. Finally, the paper assesses the significance of this research, and considers further work.

2 Formal Specification

One serious problem with many formal models of multi-agent systems is that they are very rarely applicable to building real systems. Indeed the gap between formal theoretical models on the one hand and implemented systems on the other is now a widely acknowledged issue [13, 30] and whilst

there have been several notable efforts to address this (e.g. [46, 58]), it is still the case that most new formal models do not even outline their role in system design and implementation.

There is a large number of formal techniques and languages available to specify properties of software systems [14] including state-based languages such as VDM [33], Z [53] and B [36], process-based languages such as CCS [41] and CSP [28], temporal logics [20], modal logics [10], and Statecharts [55]. However, in bringing together the need for formal models on one hand and computational models that relate to software development on the other, we adopt the Z specification language. Z is the most widely used of formal methods from software engineering, and offers arguably the best chance of the agent models developed achieving a significant degree of adoption in the broader community.

The Z language is used both in industry and academia, as a strong and elegant means of formal specification, and is supported by a large array of books (e.g. [2, 27]), articles (e.g. [3, 4]) and development tools. Its use in industry is both a reflection of its accessibility (the language is based on simple notions from first order logic and set theory) and its expressiveness, allowing a consistent, unified and structured account of a computer systems and its associated operations. Furthermore, Z is gaining increasing acceptance as a tool within the artificial intelligence community (e.g. [24, 39, 42]) and is consequently appropriate in terms of standards and dissemination capabilities.

2.1 Z Language Syntax

The key syntactic element of Z is the schema, which allows specifications to be structured into manageable modular components. The schema below has a very similar semantics to the Cartesian product of natural numbers but without any notion of order. In addition, and as can be seen in the example, the schema enables any declared variables to be constrained. In this case the schema declares two variables that are both natural numbers, and constrains them so that the variable *first* is less than or equal to the variable $second$.

$$
\begin{array}{|l}
\hline
Pair \\
\hline
first : \mathbb{N} \\
second : \mathbb{N} \\
\hline
first \leq second \\
\hline
\end{array}
$$

Modularity is facilitated in Z by allowing schemas to be included within other schemas. We can select a state variable, var, of a schema, $schema$, by writing $schema.var$. For example, it should be clear that $Pair.first$ refers to the variable *first* in the schema $Pair$.

Now, operations in a state-based specification language are defined in terms of *changes to the state*. Specifically, an operation relates variables of the state after the operation (denoted by dashed variables) to the value of the variables before the operation (denoted by undashed variables). Operations may also have inputs (denoted by variables with question marks), outputs (exclamation marks) and a precondition. In the $GettingCloser$ schema below, there is an operation with an input variable, $new?$; if $new?$ lies between the variables *first* and $second$, then the value of *first* is replaced with the value of $new?$. The original value of *first* is the output as $old!$. The $\Delta Pair$ symbol, is an abbreviation for $Pair \land Pair'$ and, as such, includes in this schema all the variables and predicates of the state of $Pair$ before and after the operation.

$$
\begin{array}{|l}
\hline
\textit{GettingCloser} \\
\textit{new}? : \mathbb{N} \\
\Delta \textit{Pair} \\
\textit{old}! : \mathbb{N} \\
\hline
\textit{first} \leq \textit{new}? \\
\textit{new}? \leq \textit{second} \\
\textit{first}' = \textit{new}? \\
\textit{second}' = \textit{second} \\
\textit{old}! = \textit{first} \\
\hline
\end{array}
$$

A key type in the specification contained in this paper is the relation type, expressing a mapping between two sets: a source set and a target set. The type of a relation with source X and target Y is a simply a set of ordered pairs, (x, y). More formally, we have the following definition for the relation type.

$$X \leftrightarrow Y == \mathbb{P}(X \times Y)$$

By way of example, consider the following relation between natural numbers.

$$\textit{examplerelation} : \mathbb{N} \leftrightarrow \mathbb{N}$$
$$\textit{examplerelation} = \{(3, 4), (5, 2), (6, 3)\}$$

The domain of a relation (dom) is the set of related source elements, while the range (ran) is the set of target elements. In the example above we have the following.

$$\text{dom } \textit{examplerelation} = \{3, 5, 6\}$$
$$\text{ran } \textit{examplerelation} = \{2, 3, 4\}$$

In the following example we show how a set of elements can be defined using set comprehension.

$$\{x : \mathbb{N} \mid x < 7 \bullet 2 * x\} = \{0, 2, 4, 6, 8, 10, 12\}$$

The "bullet" can also be used in predicates such as the one below, which states that the square of any natural number less than 10 is less than 100.

$$\forall n : \mathbb{N} \mid n < 10 \bullet n * n < 100$$

2.2 Z Extensions

In the schema given below, *mapseq* takes a function and a sequence and applies the function to each element of the sequence and *mapset* takes a function and a set and applies the function to each element of the set.

$$
\begin{array}{|l}
\hline
[X, Y] \\
\textit{mapseq} : (X \nrightarrow Y) \nrightarrow \text{seq } X \nrightarrow \text{seq } Y \\
\textit{mapset} : (X \nrightarrow Y) \nrightarrow \mathbb{P} X \nrightarrow \mathbb{P} Y \\
\hline
\forall \textit{seqs} : \text{seq } X; \ \textit{xs} : \mathbb{P} X; \ \textit{fun} : X \nrightarrow Y \bullet \\
\qquad \textit{mapseq fun seqs} = \{n : \mathbb{N} \mid n \in 1 \,..\, \#\textit{seqs} \bullet (n, \textit{fun } (\textit{seqs } n))\} \ \land \\
\qquad \textit{mapset fun xs} = \{x : X \mid x \in \textit{xs} \bullet \textit{fun } x\} \\
\hline
\end{array}
$$

We have found it useful in this specification to be able to assert that an element is optional. For example, in the specification given in this paper, whether a social agent has a model of itself or not is optional. The following definitions provide for a new type, $\mathsf{optional}\ T$, for any existing type, T.

$$\mathsf{optional}\ [X] == \{xs : \mathbb{P}\, X \mid \#\, xs \leq 1\}$$

Most other syntactic constructs in this paper are fairly standard but more complete treatments of Z can be found elsewhere [53].

3 The Agent Framework

There are four types upon which all our notions in the SMART (Structured and Modular Agents and Relationship Types) agent framework are based. The definitions within this paper will be built up using only these types; they are declared below.

$$[Attribute, Action, Goal, Motivation]$$

An *entity* is something that comprises a set of attributes, a set of actions, a set of goals and a set of motivations. The schema below has a declarative part containing four variables. First, $attributes$ is the set of features of the entity. These features are the only characteristics of the entity that are manifest. They need not be perceived by any particular entity, but must be potentially perceivable in an omniscient sense. Second, $capabilities$ is the set of actions of the entity, and is sometimes referred to as the *competence* of the entity. Next, $goals$ and $motivations$ are the sets of goals and motivations of the entity respectively. Goals are simply states of affairs to be achieved in the environment, in the traditional artificial intelligence sense, while motivations are higher-level non-derivative components characterising the nature of the agent, but are related to goals. Motivations are, however, qualitatively different from goals in that they are not describable states of affairs in the environment. For example, the motivation *greed* does not specify a state of affairs to be achieved, nor is it describable in terms of the environment, but it may (if other motivations permit) give rise to the generation of a goal to rob a bank. The distinction between the motivation of greed and the goal of robbing a bank is clear, with the former providing a reason to do the latter, and the latter specifying what must be done. Finally, the predicate part states that the entity must have a non-empty set of attributes.

Entity

$$attributes : \mathbb{P}\, Attribute$$
$$capabilities : \mathbb{P}\, Action$$
$$goals : \mathbb{P}\, Goal$$
$$motivations : \mathbb{P}\, Motivation$$

$$attributes \neq \{\}$$

An *object* is any entity that has capabilities (as well as attributes). The schema defining an object is that of an entity with the further proviso that the object has a non-empty set of capabilities.

Object

$$Entity$$

$$capabilities \neq \{\ \}$$

In our framework an *agent* is an object that is serving some purpose. That is, an agent is an instantiation of an object together with an associated goal or set of goals. The schema for an agent is simply that of an object but with the further restriction that the set of goals of an agent is not empty.

$$
\begin{array}{|l}
\hline
Agent \\
Object \\
\hline
goals \neq \{\,\} \\
\hline
\end{array}
$$

We define an $Environment$ to be a set of attributes.

$$Environment == \mathbb{P}\, Attribute$$

Next we define an interaction. An *interaction* is simply what happens when actions are performed in an environment. The effects of an interaction on the environment are determined by applying the $effectinteraction$ function in the axiom definition below to the current environment and the actions taken. This axiom definition is a global variable and is consequently always in scope. We require only one function to describe all interactions, since an action will result in the same change to an environment whether taken by an object or any kind of agent.

$$effectinteraction : Environment \rightarrow \mathbb{P}\, Action \nrightarrow Environment$$

3.1 Agent Perception and Agent Action

Perception can now be introduced. An agent in an environment may have a set of percepts available, which are the possible attributes that an agent could perceive, subject to its capabilities and current state. We refer to these as the *possible percepts* of an agent. However, due to limited resources, an agent will not normally be able to perceive all those attributes possible, and will base its actions on a subset, which we call the *actual percepts* of an agent. Indeed, some agents will not be able to perceive at all. In this case, the set of possible percepts will be empty and consequently the set of actual percepts will also be empty.

To distinguish between representations of mental models and representations of the *actual* environment, a type, $View$, is defined to be the perception of an environment by an agent. This has an equivalent type to that of $Environment$, but now physical and mental components of the same type can be distinguished.

$$View == \mathbb{P}_1\, Attribute$$

It is also important to note that it is only meaningful to consider perceptual abilities in the context of goals. Thus when considering objects without goals, perceptual abilities are not relevant. Objects respond directly to their environments and make no use of percepts even if they are available. We say that perceptual capabilities are *inert* in the context of objects.

In the schema for agent perception, $AgentPercepts$, we add further detail to the definition of agents, and so include the schema $Agent$. An agent has a set of perceiving actions, $perceivingactions$, which are a subset of the capabilities of an agent. The function, $canperceive$, determines the attributes that are potentially available to an agent through its perception capabilities. When applied, its arguments are the current environment, which contains information including the agent's location and orientation (thus constraining what can be perceived) and the agent's capabilities. The second predicate line states that those capabilities will be precisely the set of perceptual capabilities. Finally,

the function, *willperceive*, describes those attributes that are actually perceived by an agent and will always be applied to its goals.

$$
\begin{array}{|l}
\textit{AgentPercepts} \\
\hline
\textit{Agent} \\
\textit{perceivingactions} : \mathbb{P}\, \textit{Action} \\
\textit{canperceive} : \textit{Environment} \rightarrow \mathbb{P}\, \textit{Action} \nrightarrow \textit{Environment} \\
\textit{willperceive} : \mathbb{P}\, \textit{Goal} \rightarrow \textit{Environment} \rightarrow \textit{View} \\
\hline
\textit{perceivingactions} \subseteq \textit{capabilities} \\
\forall\, \textit{env} : \textit{Environment};\ \textit{as} : \mathbb{P}\, \textit{Action} \bullet \\
\qquad \textit{as} \in \mathrm{dom}(\textit{canperceive env}) \Rightarrow \textit{as} = \textit{perceivingactions} \\
\mathrm{dom}\ \textit{willperceive} = \{\textit{goals}\} \\
\hline
\end{array}
$$

Directly corresponding to the goal or goals of an agent, is an action-selection function, dependent on the goals, current environment and the actual perceptions. This is specified in *AgentAct* below, with the first predicate ensuring that the function returns a set of actions within the agent's competence. Note also that if there are no perceptions, then the action-selection function is dependent only on the environment.

$$
\begin{array}{|l}
\textit{AgentAct} \\
\hline
\textit{Agent} \\
\textit{agentactions} : \mathbb{P}\, \textit{Goal} \rightarrow \textit{View} \rightarrow \textit{Environment} \rightarrow \mathbb{P}\, \textit{Action} \\
\hline
\forall\, \textit{gs} : \mathbb{P}\, \textit{Goal};\ \textit{v} : \textit{View};\ \textit{env} : \textit{Environment} \bullet \\
\qquad (\textit{agentactions gs v env}) \subseteq \textit{capabilities} \\
\mathrm{dom}\ \textit{agentactions} = \{\textit{goals}\} \\
\hline
\end{array}
$$

3.2 Agent State

The state of an agent describes an agent currently situated in some environment and is defined as follows. This includes two variables, *posspercepts*, describing those percepts possible in the current environment, and *actualpercepts*, a subset of these which are the current (actual) percepts of the agent. These are calculated using the *canperceive* and *willperceive* functions.

$$
\begin{array}{|l}
\textit{AgentState} \\
\hline
\textit{environment} : \textit{Environment} \\
\textit{AgentPercepts} \\
\textit{AgentAct} \\
\textit{posspercepts} : \textit{View} \\
\textit{actualpercepts} : \textit{View} \\
\textit{willdo} : \mathbb{P}\, \textit{Action} \\
\hline
\textit{actualpercepts} \subseteq \textit{posspercepts} \\
\textit{perceivingactions} = \{\,\} \Rightarrow \textit{posspercepts} = \{\,\} \\
\textit{posspercepts} = \textit{canperceive environment perceivingactions} \\
\textit{actualpercepts} = \textit{willperceive goals posspercepts} \\
\textit{willdo} = \textit{agentactions goals actualpercepts environment} \\
\hline
\end{array}
$$

Lastly we specify how an agent interacts with its environment. As a result of an interaction, the environment and the $AgentState$ change.

$$
\begin{array}{|l}
\hline
AgentEnvInteract \\
\Delta AgentState \\
\hline
environment' = effectinteraction\ environment\ willdo \\
posspercepts' = canperceive\ environment'\ perceivingactions \\
actualpercepts' = willperceive\ goals\ posspercepts' \\
willdo' = agentactions\ goals\ actualpercepts'\ environment' \\
\hline
\end{array}
$$

3.3 Tropistic Agents

SMART specifies a set of generic architectures. The types, functions and schemas it contains can be applied to other systems and concepts. In order to illustrate its use in this way, and to show how the model of interaction is sufficiently general to capture most types of agents, *tropistic agents* [23] are reformulated as an example of an agent. It is one of a set of core agent architectures used by Genesereth and Nilsson to demonstrate some key issues of intelligent agent design. The activity of tropistic agents, as with reflexive agents, is determined entirely by the state of the environment in which they are situated. First, the original description of tropistic agents is summarised and then reformulated using elements of SMART.

According to Genesereth and Nilsson, the set of environmental states is denoted by S. Since agent perceptions are limited in general, it cannot be assumed that an arbitrary state is distinguishable from every other state. Perceptions thus partition S in such a way that environments from different partitions can be distinguished whilst environments from the same partition cannot. The partitions are defined by the sensory function, see, which maps environments contained in S to environments contained in T, the set of all observed environments. The effectory function, do, which determines how environments change when an agent performs an action, taken from the set of the agent actions, A, maps the agent's action and the current environment to a new environment. Finally, action-selection for a tropistic agent, $action$, is determined by perceptions and maps elements of T to elements of A. Tropistic agents are thus defined by the following tuple.

$$(S, T, A, see : S \rightarrow T, do : A \times S \rightarrow S, action : T \rightarrow A)$$

3.3.1 Reformulating Perception

The SMART framework can be applied to reformulate tropistic agents by first defining types: equating the set S to the SMART type, $Environment$; the set T (as it refers to agent perceptions), to the type $View$; and the set, A, to the type $Action$. The following type definitions can then be written.

$$S == Environment \land T == View \land A == Action$$

According to SMART, tropistic agents are not autonomous. Thus the agent-level of conceptualisation is the most suitable level, and these are the models chosen. The functions defining architecture at this level are $canperceive$, $willperceive$ and $agentactions$, defining the possible percepts, actual percepts and performed actions, respectively. The effect of actions on environments is independent of the level chosen in the agent hierarchy and defined by $effectinteraction$. Recall that these functions have the following type signatures.

$$canperceive : Environment \rightarrow \mathbb{P}\, Action \nrightarrow View$$
$$willperceive : \mathbb{P}\, Goal \rightarrow View \rightarrow View$$
$$agentactions : \mathbb{P}\, Goal \rightarrow View \rightarrow Environment \rightarrow \mathbb{P}\, Action$$
$$effectinteraction : Environment \rightarrow \mathbb{P}\, Action \rightarrow Environment$$

These functions include explicit reference to agent goals, which are not represented in the model of tropistic agents since they are implicitly fixed in the hard-coded functions. In what follows, the value of these goals is taken to be gs and accordingly set all goal parameters of SMART functions to this value.

The goals of a tropistic agent do not constrain the selection of its perceptions from those that are available, and *willperceive* is defined as the identity function on observed environments. In SMART, the perceiving actions are used at every perceiving step so that the second argument of *canperceive* is always applied to the perceiving actions (*perceivingactions*) of the agents as specified in the *AgentPerception* schema. Accordingly, tropistic perception is reformulated in the second predicate below. There is an implicit assumption that tropistic agents are capable perceivers; perceptions are always a subset of the actual environment. This assumption is formalised in the last of the three predicates below that together define tropistic perception.

$$willperceieve\ gs = \{v : View \bullet (v, v)\}$$
$$\forall e : S \bullet see\ e = willperceive\ gs\ (canperceive\ e\ perceivingactions)$$
$$\forall e, v : S \bullet willperceive\ gs\ (canperceive\ e\ perceivingactions) \subseteq e$$

The set of partitions in S can be calculated using set comprehension.

$$partitions == \{e, v : Environment \mid v = see\ e \bullet see \rhd \{v\}\}$$

3.3.2 Reformulating Action

The difference between the SMART framework and tropistic agent effectory functions is simply that the former allows for a set of actions to be performed rather than a single action.

$$\forall e : Environment;\ a : Action \bullet do\ (a, e) = effectinteraction\ e\ \{a\}$$

The action *selected* by a tropistic agent is dependent solely on its perceptions. In SMART, the actions *performed* are additionally dependent on goals and the environment. The environment can affect the performance of selected actions if, for example, an agent has incorrect or incomplete perceptions of it. By contrast it is assumed that a tropistic agent correctly perceives its static environment and performs actions that are equivalent to those selected. These assumptions mean that the environment does not affect the performance of actions once they have been selected. In order to specify this in SMART, the *Environment* parameter of *agentactions* is fixed to the empty set, and *action* is defined using *agentactions* as follows.

$$\forall v : T \bullet action\ v = agentactions\ gs\ v\ \{\}$$

Reformulating tropistic agents using SMART highlights several issues of note. First, SMART provides a more intuitive conceptualisation of an agent as an object with a purpose. Goals are hard-coded into tropistic agent actions and perception functions; they are neither *ascribed* to the agents nor are there any explicit mechanisms by which agent goals direct behaviour. Second, explicitly incorporating agent goals in SMART provides a more sophisticated design environment. It incorporates the premise

that agent goals change over time and that the selection of actions and perceptions must be adapted accordingly. Clearly, it is inefficient to have to re-write the functions defining action and perception selection every time new goals are adopted. Third, features of SMART are more generally applicable than those described for tropistic agents, and it can therefore be used to explicitly formalise any assumptions (implicit or otherwise) regarding the tropistic agent, its environment, and the interaction between them.

In this section we have constructed a formal specification that provides us with a hierarchy where all agents are objects, and all objects are entities, with the distinction between each category made precise. However, the agents that have been defined are not in themselves especially useful or interesting. Consequently, we must consider how we can refine this framework to develop definitions for other, more interesting and more varied kinds of agents. The next section describes and specifies four types of agents, each describing some subset of the agent class. These are autonomous agents, memory agents, planning agents and social agents. Each definition will arise through a refinement of the ideas and schemas presented above.

4 Classes of Agents

4.1 Autonomous Agents

Our definition of agents entails the notion that an entity is serving some purpose or, equivalently, that the entity can be ascribed some goal. However, we have not as yet considered how goals arise in the first place. In our view, goals are derivative components that are constructed in response to the needs either of the agent itself, or of some other agent. Goals can be adopted and transferred, but if goals are derivative as we claim, then there must be some entities in the world can generate or derive these goals. We define the non-derivative components from which goals are derived as motivations; goals are generated solely in response to motivations. It is this quality that defines autonomy; an agent that has a non-empty set of motivations (from which goals may be created) is an autonomous agent.

An *autonomous agent* is an agent together with an associated set of motivations.

$$
\begin{array}{|l}
\hline
AutonomousAgent \\
Agent \\
\hline
motivations \neq \{\,\} \\
\hline
\end{array}
$$

An autonomous agent is defined as an agent with motivations and some potential means of evaluating behaviour in terms of the environment and these motivations. In other words, the behaviour of the agent is determined by both external and internal factors. This is qualitatively different from an agent with goals because motivations are non-derivative and governed by internal inaccessible rules, while goals are derivative and relate directly to motivations.

Autonomous agents also perceive, but motivations, as well as goals, filter relevant aspects of the environment. In the schema below, the function $autowillperceive$ is then a more complex version of an agent's $willperceive$, but they are related — if we choose to interpret the behaviour of the agent solely in terms of its agenthood (and therefore its goals) then the $willpreceive$ representation is appropriate, and if we wish to interpret its behaviour in terms of its autonomy (and therefore its motivations as well as its goals) then the function $autowillperceive$ will be appropriate.

Nevertheless, that which an autonomous agent is *capable* of perceiving at any time is independent of its goals and motivations and we just import the definition of $canperceive$ from $AgentPercepts$.

$AutonomousAgentPercepts$

$AutonomousAgent$
$AgentPercepts$
$autowillperceive : \mathbb{P}\, Motivation \rightarrow \mathbb{P}\, Goal \rightarrow Environment \rightarrow View$

$\mathrm{dom}\, autowillperceive = \{motivations\}$

The next schema defines the action-selection function and includes the previous schema definitions for $AgentAct$ and $AutonomousAgent$. The action-selection function for an autonomous agent is produced at every instance by the motivations of the agent, and is always and only ever applied to the motivations of the autonomous agent.

$AutonomousAgentAct$

$AutonomousAgent$
$AgentAct$
$autoactions : \mathbb{P}\, Motivation \rightarrow \mathbb{P}\, Goal \rightarrow View \rightarrow Environment \rightarrow \mathbb{P}\, Action$

$\mathrm{dom}\, autoactions = \{motivations\}$

We also define the state of an autonomous agent in an environment by including the $AgentState$, $AutonomousAgentPercepts$ and $AutonomousAgentAct$ schemas.

$AutonomousAgentState$

$AutonomousAgentPercepts$
$AutonomousAgentAct$
$AgentState$

$actualpercepts = autowillperceive\ motivations\ goals\ posspercepts$
$willdo = autoactions\ motivations\ goals\ actualpercepts\ environment$

Now we can specify the operation of an autonomous agent performing its next set of actions in its current environment. Notice that while no explicit mention is made of any change in motivations, they may change in response to changes in the environment.

$AutonomousAgentEnvInteract$

$\Delta AutonomousAgentState$

$environment' = effectinteraction\ environment\ willdo$
$posspercepts' = canperceive\ environment'\ perceivingactions$
$actualpercepts' = autowillperceive'\ motivations'\ \ goals'\ posspercepts'$
$willdo' = autoactions\ motivations'\ goals'\ actualpercepts'\ environment'$

The essential feature in distinguishing autonomous agents from non-autonomous agents is the ability to generate their own goals according to their internal non-derivative *motivations.* Once goals are generated, they can subsequently be adopted by, and in order to create, other agents. We can extend the framework to show how an autonomous agent can generate goals.

In order to do so, we require a repository of known *goals*, which capture knowledge of limited and well-defined aspects of the world. These goals describe particular *states* or *sub-states* of the world with

each autonomous agent having its own such repository. An agent tries to find a way to mitigate motivations, either by selecting an action to achieve an existing goal, by reactively performing an action in direct response to motivations, or by retrieving a goal from a repository of known goals. The first two of these alternatives were addressed by the $autoactions$ function in the $AutonomousAgentAct$ schema seen earlier, while the last is considered briefly below.

In order to retrieve goals to mitigate motivations, an autonomous agent must have some way of assessing the effects of competing or alternative goals. Clearly, the goals that make the greatest positive contribution to the motivations of the agent should be selected. To do this, an autonomous agent must monitor its motivations for goal generation, and retrieve appropriate sets of goals from a repository of available known goals. We can define a function that takes a particular configuration of motivations and a set of existing goals and returns a numeric value representing the motivational effect of satisfying those goals. Then all that is necessary for goal generation is to find the set of goals in the goalbase that has a greater motivational effect than any other set of goals, and to update the current goals of the agent are updated to include the new goals.

4.2 Memory Agents

The agents considered above are very simple and rely solely on the environment, goals and motivations (if available) to determine action. We have not yet specified the way in which some agents may be able to take into account prior experience (except through any changes that arise in goals and motivations). Agents who cannot take past experience into account will be extremely limited as a result. By adding further detail to our existing definitions we now provide a description of agents with the ability to access an internal store of attributes or *memory* that can record, for example, prior experience and other relevant information. We call an agent with such an internal store a *memory agent*.

A memory agent does not necessarily record just those attributes that are currently available in the external environment, but may also store some other attributes regarding more general learned or acquired information.

$$
\begin{array}{|l}
\hline
MemoryAgent \\
Agent \\
memory : \mathbb{P}\, Attribute \\
\hline
memory \neq \{\} \\
\hline
\end{array}
$$

Thus a memory agent differs from the previous agents by having available, in addition to the external environment, an internal store of attributes, both of which contribute to forming the agent's current view of the world. Clearly, a memory agent will require certain perceiving actions in order to access the memory. In this respect it may often be useful to divide the perceiving actions of an agent into internal and external parts and, analogously, the environment may also be split into internal and external components, the internal environment being the memory.

Note that we refer to the internal set of attributes as an *environment*, rather than a $view$, since it is a *physical* store similar to the external environment. We can now refine the agent schemas to include this concept of memory as follows.

$$
\begin{array}{|l}
\hline
\textit{MemoryAgentPercepts} \\
\textit{MemoryAgent} \\
\textit{AgentPercepts} \\
\textit{internalperceivingactions} : \mathbb{P}\, \textit{Action} \\
\textit{externalperceivingactions} : \mathbb{P}\, \textit{Action} \\
\textit{memcanperceive} : (\textit{Environment} \times \textit{Environment}) \rightarrow \mathbb{P}\, \textit{Action} \nrightarrow \textit{Environment} \\
\hline
\textit{internalperceivingactions} \cup \textit{externalperceivingactions} = \textit{perceivingactions} \\
\textit{internalperceivingactions} \cap \textit{externalperceivingactions} = \{\} \\
\forall\, \textit{env}1, \textit{env}2 : \textit{Environment};\ \textit{as} : \mathbb{P}\, \textit{Action} \bullet \\
\qquad \textit{as} \in \mathrm{dom}(\textit{memcanperceive}\ (\textit{env}1, \textit{env}2)) \Rightarrow \textit{as} = \textit{perceivingactions} \\
\hline
\end{array}
$$

A memory agent's possible percepts are derived from applying its perceiving actions to both its external environment and its internal memory. Depending on its goals, the agent will select a subset of these available attributes, as defined previously by *willpreceive* in the $AgentPercepts$ schema. The action that such an agent selects at any time is also determined in the same way as defined previously in the $AgentAct$ schema, since the memory is carried through possible percepts and actual percepts to the action-selection function, *agentactions*.

$$
\begin{array}{|l}
\hline
\textit{MemoryAgentAct} \\
\textit{AgentAct} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{MemoryAgentState} \\
\textit{MemoryAgentPercepts} \\
\textit{AgentState} \\
\hline
\textit{posspercepts} = \textit{memcanperceive}\ (\textit{environment}, \textit{memory})\ \textit{perceivingactions} \\
\hline
\end{array}
$$

As a result of these refinements, we must also consider the consequences of an action on the environment and memory. The performance of some set of actions may, in addition to causing a change to the external environment, also cause a change to the memory of the agent. In this respect, we define two functions for these interactions below, where the external environment function is exactly as defined earlier, but the memory is updated as a function of both the internal environment, external environment and the current goals of the agent. Goals are relevant here because they may constrain what is recorded in memory, and what is not.

$$
\begin{array}{|l}
\textit{externaleffectinteraction} : \textit{Environment} \rightarrow \mathbb{P}\, \textit{Action} \nrightarrow \textit{Environment} \\
\textit{internaleffectinteraction} : \textit{Environment} \rightarrow \textit{Environment} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbb{P}\, \textit{Action} \nrightarrow \mathbb{P}\, \textit{Goal} \rightarrow \textit{Environment}
\end{array}
$$

We can now refine another schema to take into account these changes. The following schema, which specifies how a memory agent interacts with its environment, is a new version of the $AgentEnvInteract$ schema defined earlier.

$$
\begin{array}{|l}
\hline
MemoryAgentEnvInteract \\
\Delta MemoryAgent \\
AgentEnvInteract \\
externalenvironment : Environment \\
externalenvironment' : Environment \\
\hline
externalenvironment' = externaleffectinteraction\ externalenvironment\ willdo \\
memory' = internaleffectinteraction\ memory\ externalenvironment\ willdo\ goals \\
\hline
\end{array}
$$

These memory agents are similar in spirit to the *knowledge-level agents* of Genesereth and Nilsson [23], in which an agent's mental actions are viewed as inferences on its database, so that prior experience and knowledge can be taken into account when considering what action to take.

4.3 Planning Agents

Planning is the process of finding a sequence of actions to achieve a specified goal. Our definition of agents requires the presence of goals, and although we have briefly discussed how goals may be generated, we have not considered how agents plan to achieve them. This section increases the complexity of the agent specification so that it captures the essence of a *planning agent.*

4.3.1 Modelling Plans

This involves defining first the *components* of a plan, and then the *structure* of a plan, as shown in Figure 1. The components, which we call *plan-actions,* each consist of a *composite-action* and a set of related entities as described below. The structure of plans defines the relationship of the component plan-actions to one another. For example, plans may be *total* and define a sequence of plan-actions, *partial* and place a partial order on the performance of plan-actions, or *trees* and, for example, allow choice between alternative plan-actions at every stage in the plan's execution.

We identify four types of action that may be contained in plans, called *primitive, template, concurrent-primitive* and *concurrent-template.* There may be other categories and variations on those we have chosen, but not only do they provide a starting point for specifying systems, they also illustrate how different representations can be formalised and incorporated within the same model. A primitive action is simply a base action as defined in the agent framework, and an action template provides a high-level description of what is required by an action, defined as the set of all primitive actions that may result through an instantiation of that action-template. An example where the distinction is manifest is in dMARS (see Section 7.2), where template actions would represent action formulae containing free variables. Once all the free variables are bound to values, the action is then a primitive action and can be performed. We also define a concurrent-primitive action as a set of primitive actions to be performed concurrently and a concurrent action-template as a set of template actions that are performed concurrently. A new type, $ActnComp$, is then defined as a *compound-action* to include all four of these types.

Actions must be performed by entities, so we associate every composite-action in a plan with a set of entities, such that each entity in the set can potentially perform the action. At some stage in the planning process this set may be empty, indicating that no choice of entity has yet been made. We define a *plan-action* as a set of pairs, where each pair contains a composite-action and a set of those entities that could potentially perform the action. Plan-actions are defined as a *set of pairs* rather than a *single pair* so that plans containing simultaneous actions can be represented.

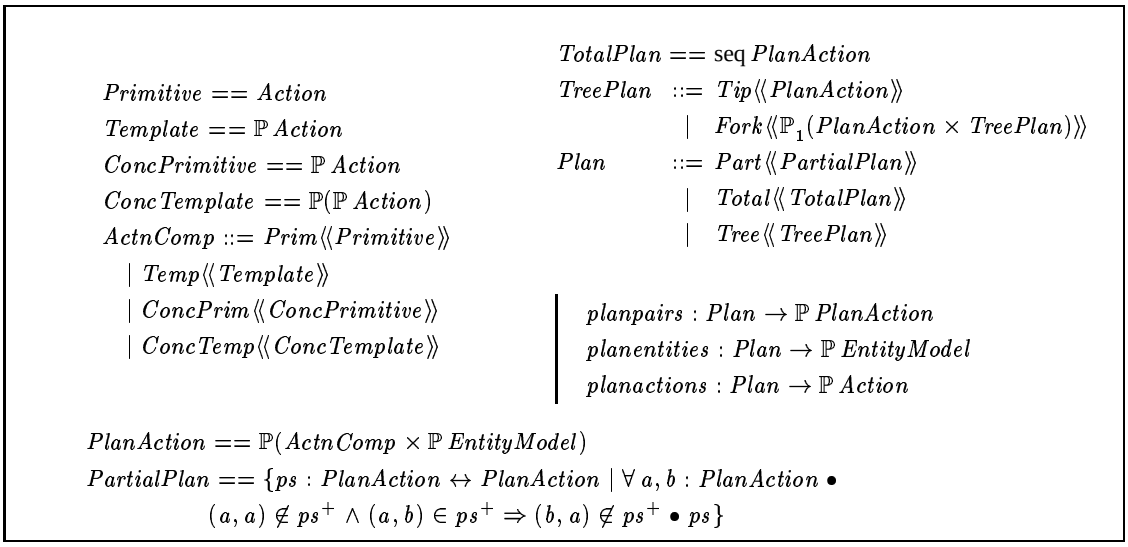

Figure 1: Plan components and structure

We specify three commonly-found categories of plan according to their structure as discussed earlier, though other types may be specified similarly.

- *Partial Plans.*

  A partial plan imposes a partial order on the execution of actions, subject to two constraints. First, an action cannot be performed before itself and, second, if plan-action a is before b, b cannot be before a. Formally, a partial plan is a relationship between plan-actions such that the pair (a, a) is not in the transitive closure and, further, if the pair (a, b) is in the transitive closure of the relation then the pair (b, a) is not.

- *Total Plans.* A plan consisting of a total order of plan-actions is a total plan. Formally, this is represented as a sequence of plan-actions.

- *Tree Plans* A plan that allows a choice between actions at every stage is a tree. In general, a tree is either a leaf node containing a plan-action, or a fork containing a node, and a (non-empty) set of branches each leading to a tree.

These are formalised in Figure 1.

Using these components we can define a *planning agent.* Any planning agent must have a set of goals currently being pursued, $goals$, and a set of plans associated with these goals, $plans$. The plans associated with each of the goals is given by the function, $activeplangoal$. There will also be a repository of *all* goals, $goalbase$, a repository of *all* plans, $planbase$, and a function associating plans in the $planbase$ with the goals in the $goalbase$, $plangoal$.

$$
\begin{array}{|l}
\hline
\textit{PlanningAgent} \\
\hline
\textit{Agent} \\
\textit{activeplangoal} : \textit{Goal} \rightarrow \mathbb{P}\, \textit{Plan} \\
\textit{plans} : \mathbb{P}\, \textit{Plan} \\
\textit{plangoal} : \textit{Goal} \rightarrow \mathbb{P}\, \textit{Plan} \\
\textit{goalbase} : \mathbb{P}\, \textit{Goal} \\
\textit{planbase} : \mathbb{P}\, \textit{Plan} \\
\hline
\mathrm{dom}\, \textit{activeplangoal} = \textit{goals} \\
\bigcup(\mathrm{ran}\, \textit{activeplangoal}) = \textit{plans} \\
\bigcup(\mathrm{ran}\, \textit{plangoal}) = \textit{planbase} \\
\mathrm{dom}\, \textit{plangoal} \subseteq \textit{goalbase} \\
\textit{goals} \subseteq \textit{goalbase} \\
\hline
\end{array}
$$

The way in which a planning agent chooses how to act is now also a function of its current plans. This is shown in the *PlanningAgentAct* schema below.

$$
\begin{array}{|l}
\hline
\textit{PlanningAgentAct} \\
\hline
\textit{PlanningAgent} \\
\textit{AgentAct} \\
\textit{planningagentactions} : \mathbb{P}\, \textit{Goal} \rightarrow \mathbb{P}\, \textit{Plan} \rightarrow \textit{View} \rightarrow \textit{Environment} \rightarrow \mathbb{P}\, \textit{Action} \\
\hline
\forall\, \textit{gs} : \mathbb{P}\, \textit{Goal};\ \textit{ps} : \mathbb{P}\, \textit{Plan};\ \textit{v} : \textit{View};\ \textit{env} : \textit{Environment} \\
\qquad \bullet\ (\textit{planningagentactions}\ \textit{gs}\ \textit{ps}\ \textit{v}\ \textit{env}) \subseteq \textit{capabilities} \\
\mathrm{dom}\, \textit{planningagentactions} = \{\textit{goals}\} \\
\forall\, \textit{ps} : \mathbb{P}\, \textit{Plan} \bullet \textit{ps} \in \mathrm{dom}(\textit{planningagentactions}\ \textit{goals}) \Rightarrow \textit{ps} = \textit{plans} \\
\hline
\end{array}
$$

Here, we have only extended our description of an agent to include the ability to plan. Further work is necessary to investigate and specify how the plans of an agent also affect its reasoning. For example, we must address the questions of when an agent should abandon plans, generate new plans, abandon goals because of a lack of appropriate plans, and so on. However, this is beyond the scope of the current work and will not be addressed further in this paper. Nevertheless, we have provided a framework within which such issues and related theories and systems can be formally presented as we shall show in the next section when we show how plans are modelled for various existing systems.

4.4 Sociological Agents

4.4.1 Agent Models

We have already stated that an agent will have certain percepts available to it. If an agent can make sense of these attributes and group certain sets of them together into entity-describing models, then we have the beginnings of a sociological agent, which we take to be an agent that is aware of other agents, and their role and function. Specifically the agent framework provides the structure that allows an agent to construct meaningful and useful models of these roles and functions in a very simple but effective way. Such models are described below.

If the agent does not have a memory, then the union of the attributes of the set of entities it models must be a subset of its current perceptions. If the agent does have a memory, however, this condition can be relaxed.

Once again, we refine the schemas given earlier to construct our model of a sociological agent. The schema below describes an agent that has grouped attributes into distinct entities.

$$
\begin{array}{|l}
\hline
AgentModelEntities \\
\hline
AgentState \\
entities : \mathbb{P}\, Entity \\
\hline
entities \neq \{\} \\
\bigcup \{e : entities \bullet e.attributes\} \subseteq actualpercepts \\
\hline
\end{array}
$$

Though the agent does not necessarily have memory, this still constitutes a model of the world that it possesses, since it imposes a structure by grouping attributes. This kind of modelling is used by mechanisms such as a robot arm on a production line. The arm is only concerned with the perceptual stimuli needed for it to perform the appropriate action on an entity. In many cases, it will not need to know about the capabilities of the entity.

Now an agent may, in addition, associate capabilities with some entity, and its model of the world will therefore be a collection entities and objects. (This will typically involve the use of memory, but we will not consider memory further here, so that we may clearly differentiate the qualities that arise for distinct reasons.)

$$
\begin{array}{|l}
\hline
AgentModelObjects \\
\hline
AgentModelEntities \\
objects : \mathbb{P}\, Object \\
\hline
objects \neq \{\} \\
objects \subseteq entities \\
\hline
\end{array}
$$

Similarly, a more sophisticated agent may be able to model the world as a set of entities, objects and agents.

$$
\begin{array}{|l}
\hline
AgentModelAgents \\
\hline
AgentModelObjects \\
agents : \mathbb{P}\, Agent \\
\hline
agents \neq \{\} \\
agents \subseteq objects \\
\hline
\end{array}
$$

At this level of modelling, an agent is aware of the concept of agenthood. That is, it is aware that some of the entities in the world are serving a purpose. However, we cannot yet claim this to be a sociological agent, since it must be aware not only that a goal is being satisfied, but also *why* the goal exists. In other words, it must know that the goal has been generated by some agent. A sociological agent must thus understand the concept of autonomous agents, which are the only agents capable of generating goals. We therefore define a *sociological agent* to be any agent that has the ability to model an entity as an autonomous agent.

Note that in building up this notion of a sociological agent, we are providing only a basic foundational concept that allows us to describe agents capable of modelling others. Further categories of agents may be constructed on top of this and, indeed, this notion of sociological agents is distinct from social agents that interact with others. However, in order for social behaviour to be effective, we argue that sociological capabilities are needed.

$$
\begin{array}{|l}
\hline
\textit{SociologicalAgent} \\
\textit{AgentModelAgents} \\
autonomousagents : \mathbb{P}\, AutonomousAgent \\
\hline
autonomousagents \neq \{\} \\
autonomousagents \subseteq agents \\
\hline
\end{array}
$$

According to this view, an agent can be sociological even it has no social capabilities (such as rhetorical devices) other than that it recognises autonomy.

If we expand the previous schema, then a sociological agent considers the world to consist of entities, objects, agents and autonomous agents, where all autonomous agents are agents, all agents are objects and all objects are entities. In addition, if it can recognise agents that are autonomous and objects that are agents then it will certainly be able to recognise agents that are not autonomous and objects that are not agents. These are known as *Server Agents* and *Neutral Objects* respectively, and are defined as follows.

$$
\begin{array}{|l}
\hline
\textit{ServerAgent} \\
\textit{Agent} \\
\hline
motivations = \{\ \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{NeutralObject} \\
\textit{Object} \\
\hline
goals = \{\ \} \land motivations = \{\} \\
\hline
\end{array}
$$

A sociological agent may also have a model of itself. This is given in the following schema. Note that a sociological agent is not necessarily an autonomous agent, nor is an autonomous agent a sociological agent. (For a definition of optional the reader is asked to consult the Appendix A.)

$$
\begin{array}{|l}
\hline
\textit{SociologicalAgentModel} \\
\textit{SociologicalAgent} \\
neutralobjects : \mathbb{P}\, NeutralObject \\
serveragents : \mathbb{P}\, ServerAgent \\
self : \mathsf{optional}\ [Agent] \\
\hline
agents = autonomousagents \cup serveragents \\
objects = neutralobjects \cup agents \\
\hline
\end{array}
$$

Naturally, if an agent want to take advantage of its ability to model these agents, it will need to use certain persuasive devices as described in [6], but which will not be considered here.

Suppose, for example, that a robot wishes to use a radio. Also, suppose that the radio is already switched on, and that the robot understands that the radio is serving some purpose. Since the robot is a sociological agent and understands autonomy, it is aware that the radio is serving some purpose, ultimately for some autonomous agent. The robot can simply take control of the radio (if it had the necessary capabilities), fully aware that the radio would not then be serving its original purpose. Alternatively, the robot may decide not to interfere with the radio. In both cases, there is no social

behaviour. However, if the robot has the ability to model the relationship between the radio and the agents for which that radio is serving a purpose, which may in turn be serving a purpose for other agents, it may be able to identify the autonomous agent at the top of the regressive agent chain. In this case, the robot may attempt to persuade this agent to release the radio from its original purpose, so that it may use it instead.

In other words, we still need an understanding of the relationships between the various entities in the world according to the purposes they serve. We need to be able to identify the originators of these purposes or goals, the autonomous agents that generated them. This requires an ability to model an agent directly *engaging* a server agent, and an autonomous agent *cooperating* with another autonomous agent. We now specify the social structures we call *engagements* and *cooperations* that exist in a multi-agent world.

4.4.2 Engagement and Cooperation

A direct engagement takes place whenever a neutral-object or a server-agent adopts some goals. In a direct engagement, an agent with some goals, which we call the *client,* uses another agent, which we call the *server,* to assist them in the achievement of those goals. Note that according to our previous definition, a server-agent is non-autonomous. It either exists already as a result of some other engagement, or is instantiated from a neutral-object for the current engagement. There is no restriction placed on a client-agent.

We define a *direct engagement* in the following schema, which consists of a client agent, $client$, a server agent, $server$, and the goal that $server$ is satisfying for $client$. Necessarily, an agent cannot engage itself, and both agents must have the goal of the engagement.

$$
\begin{array}{|l}
\hline
DirectEngagement \\
client : Agent \\
server : ServerAgent \\
goal : Goal \\
\hline
client \neq server \\
goal \in (client.goals \cap server.goals) \\
\hline
\end{array}
$$

An *engagement chain* represents a sequence of *direct engagements.* For example, suppose a robot uses a computer terminal to run a program to access a database in order to locate a library book, then there is a direct engagement between the robot and the terminal, of the terminal and the program, and of the program and the database, all with the goal of locating the book. An engagement chain thus represents the goal and all the robots involved in the sequence of direct engagements. In the above example, the agents involved would be as follows:

$$Robot, Terminal, Program, Database$$

Specifically, an *engagement chain* comprises some goal $goal$, the autonomous client-agent that generated the goal, $autoagent$, and a sequence of server-agents, $chain$, where each one in the sequence is directly engaging the next. For any engagement chain, there must be at least one server-agent, all the agents in involved must share $goal$, and the same agent cannot be involved more than once.

$$
\begin{array}{|l}
\hline
\textit{EngagementChain} \\
\hline
goal : Goal \\
autoagent : AutonomousAgent \\
chain : \text{seq}_1\ Agent \\
\hline
goal \in autoagent.goals \\
goal \in \bigcup \{ s : Agent \mid s \in \text{ran}\ chain \bullet s.goals \} \\
\#(\text{ran}\ chain) = \#chain \\
\hline
\end{array}
$$

The term *cooperation* is reserved for use only when the parties involved are autonomous and potentially capable of resisting. If they are not autonomous (and not capable of resisting), then one simply *engages* the other. A *cooperation* describes a goal, the autonomous agent that originally generated that goal, and those autonomous agents who have adopted that goal from the generating agent. Thus in this view, cooperation cannot occur unwittingly between autonomous agents.

$$
\begin{array}{|l}
\hline
\textit{Cooperation} \\
\hline
goal : Goal \\
generatingagent : AutonomousAgent \\
cooperatingagents : \mathbb{P}\ AutonomousAgent \\
\hline
\#cooperatingagents \geq 1 \\
\forall\ aa : cooperatingagents \bullet goal \in aa.goals \\
goal \in generatingagent.goals \\
\hline
\end{array}
$$

A sociological agent thus views the world as a collection of engagements, engagement chains and cooperations between the entities in the world.

$$
\begin{array}{|l}
\hline
\textit{NewSociologicalAgentModel} \\
\hline
SociologicalAgentModel \\
dengagement : \mathbb{P}\ DirectEngagement \\
engchain : \mathbb{P}\ EngagementChain \\
cooperations : \mathbb{P}\ Cooperation \\
\hline
\end{array}
$$

These schemas provide useful structure that can be exploited by intelligent agents for more effective operation. This is only possible if each agent maintains a model of their view of the world. Specifically, each agent must maintain information about the different entities in the environment, so that both existing and potential relationships between those entities may be understood and consequently manipulated as appropriate.

5 Application of the Framework

The structures described above will be present to a greater or lesser extent in all multi-agent systems. Naturally, these models and the models that an agent has of other entities can become even more sophisticated. For example, an agent may model other agents as planning agents as we shall see later in this section. In this way, agents can coordinate their activities and enlist the help of others in order that plans can be achieved successfully and efficiently.

In this section, we complete the path from our initial framework through models of varying levels of abstraction to detailed formal specifications of three distinct applications. The first is dMARS (the

distributed Multi-Agent Reasoning System), which has been applied in perhaps the most significant multi-agent applications to date. The second is the well-known contract net protocol [51, 52, 12], which again is situated in the domain of practical implemented systems. The third application is the social dependence network[49, 50], which is a structure that forms the basis of a computational model of Social Power Theory[7, 8]. These networks allow agents to reason about and understand the collective group of agents that make up the multi-agent world in which they operate. Below, we consider each of these in turn, and show how they can be formalised in the context of previous models. In order to achieve this, we reuse, refine and elaborate the schemas presented so far, in order to specify these multi-agent systems at a detailed level of description.

5.1 Application 1: The distributed Multi-Agent Reasoning System (dMARS)

While many different and contrasting single-agent architectures have been proposed, perhaps the most successful are those based on the belief-desire-intention (BDI) framework. In particular, the Procedural Reasoning System (PRS), has progressed from an experimental LISP version to a full C++ implementation known as the distributed Multi-Agent Reasoning System (dMARS). PRS, which has its conceptual roots in the belief-desire-intention (BDI) model of practical reasoning, has been the subject of a dual approach by which a significant commercial system has been produced while the theoretical foundations of the BDI model continue to be closely investigated.

As part of our work, we have sought to formalise these BDI systems through the direct representation of the implementations on the one hand, and through refinement of the detailed models constructed through the abstract agent framework on the other. This work has included the formal specification [16] of the AgentSpeak(L) language developed by Rao [46], which is a programming language based on an abstraction of the PRS architecture; irrelevant implementation detail is removed, and PRS is stripped to its bare essentials. Our specification reformalises Rao's original description so that it is couched in terms of state and operations on state that can be easily refined into an implemented system. In addition, being based on a simplified version of dMARS, the specification provides a starting point for actual specifications of these more sophisticated systems. Subsequent work continued this theme by moving to produce an abstract formal specification of dMARS itself, through which an operational semantics for dMARS was provided, offering a benchmark against which future BDI systems and PRS-like implementations can be compared.

Due to space constraints, we cannot hope to get anywhere near a specification of either of these systems, but instead we aim to show how we can further refine the models of plans described above to get to a point at which we can specify the details of such implementations. The value of this is in the ease of comparison and analysis with the more abstract notions described earlier.

We begin our specification, shown in Figure 2 by defining the allowable *beliefs* of an agent in dMARS, which are like PROLOG facts. To start, we define a *term*, which is either a variable or a function symbol applied to a (possibly empty) sequence of terms, and an *atom*, a predicate symbol applied to a (possibly empty) sequence of terms. In turn, a *belief formula* is either an atom or the negation of an atom, and the set of *beliefs* of an agent is the set of all ground belief formulae (i.e. those containing no variables). (We assume an auxiliary function $belvars$ which, given a belief formula, returns the set of variables it contains.) Similarly, a situation formula is an expression whose truth can be evaluated with respect to a set of beliefs. A goal is then a belief formula prefixed with an achieve operator or a situation formula prefixed with a query operator. Thus an agent can have a goal either of achieving a state of affairs or of determining whether the state of affairs holds.

The types of action that agents can perform may be classified as either *external* (in which case the domain of the action is the environment outside the agent) or *internal* (in which case the domain of the

$[Var, FunSym, PredSym]$

$Term ::= var\langle\langle Var \rangle\rangle \mid functor\langle\langle FunSym \times \text{seq } Term \rangle\rangle$

Atom

$head : PredSym$

$terms : \text{seq } Term$

$BeliefFormula ::= pos\langle\langle Atom \rangle\rangle \mid not\langle\langle Atom \rangle\rangle$

$Belief == \{b : BeliefFormula \mid belvars\ b = \varnothing \bullet b\}$

$SituationFormula ::= belform\langle\langle BeliefFormula \rangle\rangle$
$\quad \mid and\langle\langle SituationFormula \times SituationFormula \rangle\rangle$
$\quad \mid or\langle\langle SituationFormula \times SituationFormula \rangle\rangle$
$\quad \mid true \mid false$

$Goal ::= achieve\langle\langle BeliefFormula \rangle\rangle \mid query\langle\langle SituationFormula \rangle\rangle$

ExtAction

$name : ActionSym$

$terms : \text{seq } Term$

$IntAction ::= add\langle\langle BeliefFormula \rangle\rangle \mid remove\langle\langle BeliefFormula \rangle\rangle$

$TriggerEvent ::= addbelevent\langle\langle Belief \rangle\rangle$
$\quad \mid rembelevent\langle\langle Belief \rangle\rangle$
$\quad \mid toldevent\langle\langle Atom \rangle\rangle$
$\quad \mid goalevent\langle\langle Goal \rangle\rangle$

$Branch ::= extaction\langle\langle ExtAction \rangle\rangle \mid intaction\langle\langle IntAction \rangle\rangle \mid subgoal\langle\langle Goal \rangle\rangle$

$Body ::= End\langle\langle State \rangle\rangle \mid Fork\langle\langle \mathbb{P}_1(State \times Branch \times Body) \rangle\rangle$

Plan

$inv : TriggerEvent$

$context : \text{optional } [SituationFormula]$

$body : Body$

$maint : SituationFormula$

$succ : \text{seq } IntAction$

$fail : \text{seq } IntAction$

Figure 2: Plans in dMARS

action is the agent itself). External actions are specified as if they are procedure calls, and comprise an external action symbol (analogous to the procedure name) taken from the set $[ActionSym]$, and a sequence of terms (analogous to the parameters of the procedure). Internal actions may be one of two types: add or remove a belief from the data base.

Plans are *adopted* by agents and, once adopted, constrain an agent's behaviour and act as *intentions*. They consists of six components: an *invocation condition* (or *triggering event*); an optional *context* (a situation formula) that defines the pre-conditions of the plan, i.e., what must be believed by the agent for a plan to be executable; the *plan body*, which is a tree representing a kind of flow-graph of actions to perform; a *maintenance condition* that must be true for the plan to continue executing; a set of *internal actions* that are performed if the plan succeeds; and finally, a set of *internal actions* that are performed if the plan fails. The tree representing the body has states as nodes, and arcs (branches) representing either a goal, an internal action or an external action as defined below. Executing a plan successfully involves traversing the tree from the root to any leaf node.

A trigger event causes a plan to be adopted, and four types of events are allowable as triggers: the acquisition of a new belief; the removal of a belief; the receipt of a message; or the acquisition of a new goal. This last type of trigger event allows goal-driven as well as event-driven processing. As noted above, plan bodies are trees in which arcs are labelled with either goals or actions and states are place holders. Since states are not important in themselves, we define them using the given set $[State]$. An arc (branch) within a plan body may be labelled with either an internal or external action, or a subgoal. Finally, a dMARS plan body is either an *end tip* containing a state, or a *fork* containing a state and a non-empty set of branches each leading to another tree.

All these components can then be brought together into the definition of a plan. The basic execution mechanism for dMARS agents involves an agent matching the trigger and context of each plan against the chosen event in the event queue and the current set of beliefs, respectively, and then generating a set of candidate, matching plans, selecting one, and making a *plan instance* for it.

Space constraints prohibit going into further details of the various aspects of this work, but we hope that it has been possible to show how increasing levels of analysis and detail enable transition between abstract conceptual infrastructure and implemented system.

5.2 Application 2: The Contract Net Protocol

The Contract Net as described by Smith [12, 51, 52] is a collection of nodes that cooperate in achieving goals which, together, satisfy some high-level goal or task. Each node may be either a *manager* who monitors task execution and processes the results, or a *contractor* who performs the actual execution of the task.

5.2.1 The Contract Net Structure

First, we specify the different kinds of entity from which a contract net is constructed, and which participate in it. A node in a contract net is just an object.

$$CNode \quad Object$$

Then, reusing our definition of agents, we define a $ContractAgent$ as any node which is currently serving some purpose.

$$
\begin{array}{|l}
\hline
ContractAgent \\
\hline
CNode \\
Agent \\
\hline
\end{array}
$$

Davis and Smith[12] also describe a single processor node in a distributed sensing example called a *monitor* node, which starts the initialisation as the first step in net operation. If this is just a node that passes on information to another, then it is no different to the manager specified above. If it generated the goal or task to perform by itself, then it is an autonomous agent.

$$
\begin{array}{|l}
\hline
Monitor \\
\hline
AutonomousAgent \\
ContractAgent \\
\hline
\end{array}
$$

The contract net consists of $nodes$, which are objects. Of these, some are $contractagents$, which are agents, and a subset of these are $monitors$, which are autonomous agents.

$$
\begin{array}{|l}
\hline
AllNodes \\
\hline
nodes : \mathbb{P}\, Object \\
contractagents : \mathbb{P}\, Object \\
monitors : \mathbb{P}\, AutonomousAgent \\
\hline
monitors \subseteq contractagents \subseteq nodes \\
\hline
\end{array}
$$

However, this is precisely the same as the schema, $SociologicalAgent$, which described a sociological agent's view of the world. We can therefore define $AllNodes$ in terms of this schema by renaming or substituting the various state variables within it as follows.

$$
\begin{array}{|l}
\hline
AllNodes \\
\hline
SociologicalAgent[nodes/objects, contractagents/agents, monitors/autonomousagents] \\
\hline
\end{array}
$$

A manager engages contractors to perform certain tasks, where a task is just the same as a goal, since it specifies a state of affairs to be achieved.

$$Task == Goal$$

In a contract net, a $contract$ comprises a task, and a pair of nodes, a manager and a contractor. Yet again we can define a contract by reusing previous schemas. A contract is a specific type of *direct engagement* in which the client of the engagement is the manager of the contract, the server is the contractor, and the goal of the engagement is the task of the contract. Consequently, we define a contract as follows.

$$
\begin{array}{|l}
\hline
Contract \\
\hline
DirectEngagement[manager/client, contractor/server, task/goal] \\
\hline
\end{array}
$$

Now we can define the set of all contracts currently in operation in the contract net. The schema below includes $AllNodes$, and defines $contracts$ to be the set of all contracts currently in the net. The managers are the set of nodes managing a contract and the contractors are the set of nodes contracted. The union of the contractors and the managers gives the set of contract agents.

$$
\begin{array}{|l}
\hline
AllContracts \\
\hline
AllNodes \\
contracts : \mathbb{P}\, Contract \\
managers : \mathbb{P}\, ContractAgent \\
contractors : \mathbb{P}\, ContractAgent \\
\hline
managers = \{c : Contract \mid c \in contracts \bullet c.manager\} \\
contractors = \{c : Contract \mid c \in contracts \bullet c.contractor\} \\
managers \cup contractors = contractagents \\
\hline
\end{array}
$$

We also introduce the notion of eligibility. A node is eligible for a task if its actions and attributes satisfy the task requirements. *Eligibility* is a type comprising a set of actions and attributes representing an eligibility specification. This has just the same type as an object.

$$Eligibility == Object$$

The first step in establishing a contract is a *task announcement.* A $TaskAnnouncement$ is issued by a $Sender$ to a set of $Recipient$s to request bids for a particular $Task$ from agents with a given $Eligibility$ specification.

$$
\begin{array}{l}
Sender == CNode \\
Recipient == CNode
\end{array}
$$

$$
\begin{array}{|l}
\hline
TaskAnnouncement \\
\hline
sender : Sender \\
recs : \mathbb{P}\, Recipient \\
task : Task \\
eligibility : Eligibility \\
\hline
\end{array}
$$

Notice that the combination of a task together with an eligibility is, in fact, an $agent$ requirement.

In response to a task announcement, agents can evaluate their interest using *task evaluation procedures* specific to the problem at hand. If there is sufficient interest, then that agent will submit a bid to undertake to perform the task. A bid involves a node that describes a subset of itself in response to an eligibility specification, which will be used in evaluating the bid.

$$
\begin{array}{|l}
\hline
Bid \\
\hline
cnode : CNode \\
eligibility : Eligibility \\
\hline
eligibility.capabilities \subseteq cnode.capabilities \\
eligibility.attributes \subseteq cnode.attributes \\
\hline
\end{array}
$$

The state of the contract net can now be represented as the current set of nodes, contracts, task announcements and bids. Each task announcement will have associated with it some set of bids, which are just eligibility specifications as described above. In addition, each node has a means of deciding whether it is capable of, and interested in, performing certain tasks (and so bidding for them).

$$
\begin{array}{|l}
\hline
ContractNet \\
AllContracts \\
bids : TaskAnnouncement \pfun \mathbb{P}\, Bid \\
interested : CNode \to Task \to bool \\
taskannouncements : \mathbb{P}\, TaskAnnouncement \\
\hline
taskannouncements = \mathrm{dom}\, bids \\
\hline
\end{array}
$$

5.2.2 Making Task Announcements and Bids

The operation of a node making a task announcement is then given in the schema below where there is a change to $ContractNet$, but no change to $AllContracts$. A node that issues a task announcement must be an agent. Note that the variables with a ? suffix indicate *inputs*. The second part of the schema specifies that the recipients and the sender must be nodes, that the task must be in the goals of the sender, and that the sender must not be able to satisfy the eligibility requirements of the task alone. Finally, the task announcement is added to the set of all task announcements, and an empty set of bids is associated with it.

$$
\begin{array}{|l}
\hline
MakeTaskAnnouncement \\
\Delta ContractNet \\
\Xi AllContracts \\
m? : ContractAgent \\
taskann? : TaskAnnouncement \\
\hline
m? \in nodes \\
taskann?.recs \subseteq nodes \\
taskann?.sender = m? \\
taskann?.task \in m?.goals \\
\neg\, ((taskann?.eligibility.capabilities \subseteq m?.capabilities) \land \\
\qquad (taskann?.eligibility.attributes \subseteq m?.attributes)) \\
taskannouncements' = taskannouncements \cup \{taskann?\} \\
bids' = bids \cup \{(taskann?, \{\})\} \\
\hline
\end{array}
$$

In response to a task announcement, a node may make a bid. The schema below specifies that a node making a bid must be one of the receivers of the task announcement, that it must be eligible for the task, that it is interested in performing the task, and that it is not the sender. As a result of a node making a bid, the set of task announcements does not change, but the bids associated with the task announcement are updated to include the new bid.

MakeBid

$\Delta ContractNet$
$con? : CNode$
$bid? : Bid$
$ta? : TaskAnnouncement$

$bid?.cnode = con?$
$con? \in nodes$
$ta? \in taskannouncements$
$con? \in ta?.recs$
$ta?.eligibility.capabilities \subseteq bid?.eligibility.capabilities$
$ta?.eligibility.attributes \subseteq bid?.eligibility.attributes$
$interested\ con?\ (ta?.task) = True$
$con? \neq ta?.sender$
$taskannouncements' = taskannouncements$
$bids' = bids \oplus \{(ta?, bids\ ta? \cup \{bid?\})\}$

5.2.3 Making and Breaking Contracts

After receiving bids, the issuer of a task announcement awards the contract to the highest rated bid. The node that makes the award must be the node that issued the task announcement, and the bid that is selected must be in the set of bids associated with the task announcement. In order to choose the best bid, the *Rating* function is used to provide a natural number as an evaluation of a bid with respect to a task announcement. Thus the bid with the highest rating is selected. After making an award, the set of all contracts is updated to include a new contract for the particular task with the issuer of the task announcement as manager and the awarded bidder as contractor, where the contractor is instantiated from the old node as a new agent with the additional task of the contract. Notice that the contractor was previously either a neutral object in which case it now becomes instantiated as a contract agent, or a contract agent and now becomes instantiated as a *new* contract agent. The task announcement is now satisfied and removed from the system, and the set of bids is updated accordingly.

MakeAward

$\Delta ContractNet$
$m? : ContractAgent$
$ta? : TaskAnnouncement$
$bid? : Bid$
$Rating : TaskAnnouncement \rightarrow Bid \rightarrow \mathbb{N}$

$m? = ta?.sender$
$bid? \in bids\ ta?$
$\forall b : Bid \mid b \in bids\ ta? \bullet Rating\ ta?\ bid? \geq Rating\ ta?\ b$
$contracts' = contracts \cup$
$\quad \{makecontract\ ta?.task\ m?\ (NewAgent\ bid?.cnode\ ta?.task)\}$
$contractagents' = contractagents \setminus$
$\quad \{bid?.cnode\} \cup \{NewAgent\ bid?.cnode\ ta?.task\}$
$taskannouncements' = taskannouncements \setminus \{ta?\}$
$bids' = bids \setminus \{(ta?, bids\ ta?)\}$

The functions $makecontract$ and $NewAgent$ are defined as follows.

$$makecontract : Task \pfun ContractAgent \pfun CNode \pfun Contract$$

$$\begin{aligned}
&\forall\, t : Task;\ m : ContractAgent;\ c : CNode;\ con : Contract \bullet \\
&\quad makecontract\ t\ c\ m = con \Leftrightarrow t \in (m.goals) \land m \neq c \land \\
&\qquad con.task = t \land con.manager = m \land con.contractor = NewAgent\ c\ t
\end{aligned}$$

$$NewAgent : Object \to Goal \to ServerAgent$$

$$\begin{aligned}
&\forall\, g : Goal;\ old : Object;\ new : ServerAgent \bullet \\
&\quad NewAgent\ old\ g = new \Leftrightarrow new.goals = old.goals \cup \{g\} \\
&\qquad \land\ new.capabilities = old.capabilities \land new.attributes = old.attributes
\end{aligned}$$

Finally, a manager can terminate a contract where the contract is removed from the set of all contracts. Whilst the contractor will remove the task from its set of goals the manager will not, since it may still be a contractor for that task or the monitor of the goal. The goal is therefore removed only from the goals of the contractor agent. If this node is still an agent, there will be no change to $contractagents$, but if the node previously had only one goal then it will be removed from $contractagents$ since it is no longer an agent.

5.3 Application 3: Social Dependence Networks

As stated above, dependence networks are structures that form the basis of a computational model of Social Power Theory. They allow agents to reason about, and understand, the collective group of agents that make up the multi-agent world in which they operate. This section introduces dependence networks and external descriptions, data structures used to store information about other agents, based on the work reported by Sichman et al. [49].

External descriptions store information about other agents, and comprise a set of goals, actions, resources and plans for each such agent. The goals are those an agent wants to achieve, the actions are those an agent is able to perform, the resources are those over which an agent has control, and the plans are those available to the agent, but using actions and resources that are not necessarily owned by the agent. This means that one agent may *depend* on another in terms of actions or resources in order to execute a plan.

First, we briefly describe the original work, and then reformulate it in our framework. The following description and reformulation is based on work previously presented in [17].

An agent i is denoted by ag_i, and any such agent has a set of *external descriptions* of all of the other agents in the world, denoted by

$$Ext_{ag_i} \stackrel{\text{def}}{=} \bigcup_{j=1}^{n} Ext_{ag_i}(ag_j)$$

where

$$Ext_{ag_i}(ag_j) \stackrel{\text{def}}{=} \{G_{ag_i}(ag_j), A_{ag_i}(ag_j), R_{ag_i}(ag_j), P_{ag_i}(ag_j)\}$$

such that
$G_{ag_i}(ag_j)$ is the set of goals
$A_{ag_i}(ag_j)$ is the set of actions
$R_{ag_i}(ag_j)$ is the set of resources

$P_{ag_i}(ag_j)$ is the set of plans
that agent i believes agent j has.

Notice that an agent has a model of itself as well as others. The authors adopt what they call the *hypothesis of external description compatibility,* which states that any two agents will have precisely the same external description of any other agent. This is stated as follows.

$$Ext_{ag_i}(ag_i) = Ext_{ag_j}(ag_i) \ \land \ Ext_{ag_i}(ag_j) = Ext_{ag_j}(ag_j)$$

Now, $P_{ag_i}(ag_j, g_k)$ represents the *set* of plans that agent i believes that agent j has in order to achieve the goal g_k. Each plan within this set is given by $p_{ag_{i_l}}$:

$$p_{ag_{i_l}}(ag_j, g_k) \stackrel{\text{def}}{\equiv} \{g_k, R(p_{ag_{i_l}}(ag_j, g_k)), I(p_{ag_{i_l}}(ag_j, g_k))\}$$

where $R(p_{ag_{i_l}})$ represents the set of resources required for the plan and $I(p_{ag_{i_l}})$ is a *sequence* of instantiated actions used in this plan. Each instantiated action within a plan is defined by the action itself and the set of resources used in the instantiation of this action:

$$i_m(p_{ag_{i_l}}(ag_j, g_k)) \stackrel{\text{def}}{\equiv} \{a_m, R_{a_m}(p_{ag_{i_l}}(ag_j, g_k))\}$$

5.3.1 Reformulating Dependence Networks

We can easily reformulate this work in our framework. In the theory of social dependence networks a plan is taken to be a total order on primitive actions.

$$SDNplan == \text{seq}\, Action$$

We take a resource to be some entity — an object, agent or autonomous agent.

The definition of a planning agent for this work is then a simplified version of the general model given earlier.

SDNplanningAgent

$Agent$
$plans : \mathbb{P}\, SDNplan$
$planforgoal : Goal \rightarrow \mathbb{P}\, SDNplan$
$planbase : \mathbb{P}\, SDNplan$
$goalbase : \mathbb{P}\, Goal$
$passiveplanforgoal : Goal \rightarrow \mathbb{P}\, SDNplan$

$goals \subseteq \text{dom}\, planforgoal$
$\bigcup(\text{ran}\, planforgoal) = plans$
$goalbase \subseteq \text{dom}\, passiveplanforgoal$
$\bigcup(\text{ran}\, planforgoal) = planbase$

At this point we are now in a position to specify an *external description.* In order to do this, we must refine this definition of a planning agent by including three additional variables. The first, $ownedresources$, represents the set of resources an agent $owns$. The second, $instsreq$, models the set of resources needed to instantiate an action within a plan. The third, redundant variable, $resourcesofplan$, is included for readability and records the total set of resources required by a plan.

There are two predicates in the lower part of the schema, which relate the variables in the schema as follows: stripping the set of entities away from each instantiated action gives the original plan; and the resources of a plan are the union of each of the sets of entities associated with each action of the plan.

$$
\begin{array}{|l}
\hline
\textit{ExternalDescription} \\
\hline
SDNplanningAgent \\
ownedresources : \mathbb{P}\, Entity \\
instsreq : SDNplan \rightarrow (\text{seq}(Action \times \mathbb{P}\, Entity)) \\
resourcesofplan : SDNplan \rightarrow \mathbb{P}\, Entity \\
\hline
plans = mapset\ (mapseq\ first)\ (\text{ran}\ instsreq) \\
\forall\, p : SDNplan \bullet resourcesofplan\ p = \bigcup(\text{ran}(mapseq\ second\ (instsreq\ p))) \\
\hline
\end{array}
$$

Now, since every external description of an agent is the same, we can model the formalism very simply. An agent, A, has associated with it an external description, which is precisely the model that every agent (including agent A) has of agent A (according to the hypothesis of external description compatibility).

$$
\begin{array}{|l}
\hline
\textit{ExtDes} \\
\hline
extdes : Agent \pfun ExternalDescription \\
\hline
\end{array}
$$

Presenting the model within the formal framework highlights some apparent difficulties with the original formalism. First, the distinction between a resource and an agent is not clear. This is an important distinction since the nature of a plan assumes that all of the *resources* of an action have already been identified, but the agents that could possibly perform some action have not. Second, it is limiting in its representation of plans since simultaneous actions cannot be represented. In the multi-agent world this is particularly limiting because no two agents can then perform the same action simultaneously. Third, the notion of *ownership* in these external descriptions is not clear. Presumably, we should take it to mean that an agent *owns* another entity, if, for whatever the reason, that entity can be used for *any* action within its capabilities whenever the agent requires it. This is too strong, since it becomes impossible to represent the no notion of a shared resource, and clearly, a much richer notion of ownership is required in general. Finally, the *hypothesis of external description compatibility* ensures that any two agents will agree on the model of themselves and each other. However, a truly autonomous agent will have its own view of the world around it, which may bear no relation to another agent's interpretation of its world, and can never know the plans and goals of another agent; it may only infer them by evaluating the behaviour of the other agent.

In response to these problems, we use the SMART framework described earlier to ensure that an autonomous agent will have their own model of the world and, in addition, we allow for plans containing concurrent actions. The agent hierarchy allows us to be much clearer about the nature of the social relationships — such as ownership — which will depend on the types of entities and the goal dependence networks that exist between entities in the environment. Further, using the hierarchy, we do not have to arbitrarily distinguish agents from resources, but instead consider agents with different functionalities. In this way we can provide a clearer and more intuitive representation of the social structures in the world since a planning agent would have to consider merely the set of *agents* that are required in a plan.

An agent ag_i will be aaut for a given goal g_k, *according to a set of plans* P_{qk}
if there is a plan that achieves this goal in this set and every action appearing in this plan belongs to $A(ag_i)$:

An agent ag_i will be raut for a given goal g_k, *according to a set of plans* P_{qk}
if there is a plan that achieves this goal in this set and every resource appearing in this plan belongs to $R(ag_i)$:

An agent ag_i will be saut for a given goal g_k, *according to a set of plans* P_{qk}
if he is both aaut and raut for this goal.

Table 1: Original Definition of Action and Resource Autonomy

$$a_{aut}(ag_i, g_k, P_{qk}) \stackrel{\text{def}}{\equiv} \exists\, g_k \in G(ag_i)\, \exists\, p_{lk} \in P_{qk}\, \forall\, i_m \in (p_{lk}) \in I(p_{lk})\, a_m \in A(ag_i) \quad (9.6)$$

$$r_{aut}(ag_i, g_k, P_{qk}) \stackrel{\text{def}}{\equiv} \exists\, g_k \in G(ag_i)\, \exists\, p_{lk} \in P_{qk}\, \forall\, r_m \in R(p_{lk}) r_m \in R(ag_i) \quad (9.7)$$

$$s_{aut}(ag_i, g_k, P_{qk}) \stackrel{\text{def}}{\equiv} a_{aut}(ag_i, g_k, P_{qk}) \wedge r_{aut}(ag_i, g_k, P_{qk}) \quad (9.8)$$

Table 2: Original Formalisation of Action and Resource Autonomy

5.3.2 Definitions of Autonomy

Using external descriptions, Sichman et al. distinguish three distinct categories of autonomy referred to as aaut, raut and saut. According to these definitions agents are autonomous if they have the necessary capabilities and resources to achieve a goal and so do not need the help of others.

The original definitions and their formal representations as presented by Sichman at al. can be found in Table 1 and Table 2, respectively. However, there are a number of difficulties with them. First, the textual definitions are slightly deceptive since P_{qk} is an abbreviation of $P_{ag_i}(ag_q, g_k)$ and represents a *very specific* set of plans rather than any set of plans. It represents the set of plans that ag_i believes that agent ag_q has to achieve the goal g_q. Further, since *every* plan in this set necessarily achieves g_k, the textual definition includes unnecessary redundancy. All that is necessary is that P_{qk} is non-empty.

The definition of the category saut is also deceptive. An agent is in this category if, within the set of plans being analysed, there is one plan that contains actions within the agent's capabilities, and another plan that involves resources all owned by the agent. However, there is no stipulation that these plans are the same. In other words, an agent can be saut for a goal, and still not be able to achieve it, since there may be no *specific* plan that requires just the capabilities *and* resources of that agent. In this situation, all plans would then involve either the resources or the capabilities of other agents, so it makes little sense to say that the agent is autonomous with respect to this goal when it necessarily depends on others.

To address these concerns we provide slightly altered textual definitions that relate more strongly to the original SDN formalisms. An agent is *a-autonomous* for a given goal according to a set of plans of another to bring about that goal if there is a plan in this set that achieves the goal, and every action in this plan belongs to the capabilities of the agent. An agent is *r-autonomous* for a given goal according to a set of plans of another to bring about that goal if there is a plan in this set that achieves

the goal, and every resource required by the plan is owned by the agent. An agent is *s-autonomous* for a given goal if it is both *a-autonomous* and *r-autonomous.*

In the following schema, we define these three classes of autonomy using a new relation, achieves. The predicate, achieves (a, g, ps), holds precisely when an agent, a, has goal, g, and the non-empty set of plans associated with g in order to achieve it, is ps.

Thus in the schema below, the first predicate states that an agent, a, is *a-autonomous* with respect to some set of plans, ps, if and only if there is some agent, c, with goal, g, and plans, ps, to achieve g such that some plan, p in ps, contains actions all in the capabilities of a. Similar predicate are specified for *r-autonomous* and *s-autonomous.* Finally, the achieves predicate is specified as described above.

AutonomyRelations

$$
\begin{array}{l}
ExtDes \\
\mathsf{aaut}\ \_,\ \mathsf{raut}\ \_,\ \mathsf{saut}\ \_,\ \mathsf{achieves}\ \_ : \mathbb{P}(Agent \times Goal \times \mathbb{P}\, Plan) \\
\hline
\forall\, a : Agent;\ g : Goal;\ ps : \mathbb{P}\, Plan \bullet \\
\quad \mathsf{aaut}\ (a, g, ps) \Leftrightarrow \\
\qquad (\exists\, c : Agent \bullet\ \ \mathsf{achieves}\ (c, g, ps)) \wedge \\
\qquad (\exists\, p : ps \bullet (\mathrm{ran}\, p \subseteq (extdes\ a).capabilities)) \wedge \\
\quad \mathsf{raut}\ (a, g, ps) \Leftrightarrow \\
\qquad (\exists\, c : Agent \bullet\ \ \mathsf{achieves}\ (c, g, ps)) \wedge \\
\qquad (\exists\, p : ps \bullet ((extdes\ a).resourcesofplan\ p \subseteq \\
\qquad\qquad (extdes\ a).ownedresources)) \wedge \\
\quad \mathsf{saut}\ (a, g, ps) \Leftrightarrow \\
\qquad \mathsf{aaut}\ (a, g, ps) \wedge\ \ \mathsf{raut}\ (a, g, ps) \wedge \\
\quad \mathsf{achieves}\ (a, g, ps) \Leftrightarrow \\
\qquad g \in (extdes\ a).goals \wedge \\
\qquad (g, ps) \in (extdes\ a).planforgoal \wedge \\
\qquad ps \neq \{\ \}
\end{array}
$$

Consider the definition of achieves given above and in particular, the expression $g \in (extdes\ a).goals$. This states that an agent can only reason with respect to a set of plans associated with a *current* goal (i.e. one that it desires). However, it is not clear whether this relation is with respect to goals the agent desires, or to its goal base in which case the predicate would read $g \in (extdes\ a).goalbase$. Formalising such notions within the framework allows us to isolate such ambiguities.

According to these definitions, if agents are autonomous, then they may not *depend,* for resources or actions, on other agents. Consequently, the fact that a pocket calculator has the resources and the actions necessary for adding some numbers makes it autonomous. This is in marked contrast to our own definition which insists that an autonomous agent should be able to *generate* its own goals.

5.3.3 Dependence Relations

Now we can consider the types of dependencies that exist between agents. An agent, A, *a-depends* on another agent, B, for a given goal, g, according to some set of plans of another to achieve g, if it has g as a goal, is not *a-autonomous* for g, and at least one action used in this plan is in B's capabilities. An agent, A, *r-depends* on another agent, B, for a given goal, g, according to some set of plans of another to achieve g, if it has g as a goal, is not *r-autonomous* for g, and at least one instantiation used

in this plan is owned by B. An agent, A, *s-depends* on another agent, B, for a given goal, g, if it either *r-depends* or *a-depends* on B.

The first predicate in the schema below states that given two agents, a and b, a goal, g, and a set of plans according to which a is not *a-autonomous* with respect to g, a *a-depends* on b for g with respect to ps, if and only if there is some agent, c, with the goal, g, and plans to achieve g, ps, such that no plan in ps has an action in the capabilities of agent b.

$$
\begin{array}{|l}
\hline
DependencyRelations \\
AutonomyRelations \\
\mathbb{P}(Agent \times Agent \times Goal \times \mathbb{P}\, Plan) \\
\hline
\forall\, a, b : Agent;\ g : Goal;\ ps : \mathbb{P}\, Plan \mid \\
\quad a \neq b \land (g \in (extdes\ a).goals) \bullet \\
\qquad \mathsf{adep}\ (a, b, g, ps) \Leftrightarrow \\
\qquad\quad \neg\ \ \mathsf{aaut}\ (a, g, ps) \land \\
\qquad\quad (\exists\, c : Agent \bullet \\
\qquad\qquad \mathsf{achieves}\ (c, g, ps) \land \\
\qquad\qquad (\bigcup \{p : ps \bullet \mathrm{ran}\, p\} \cap \\
\qquad\qquad\quad (extdes\ b).capabilities \neq \{\})) \land \\
\qquad \mathsf{rdep}\ (a, b, g, ps) \Leftrightarrow \\
\qquad\quad \neg\ \ \mathsf{raut}\ (a, g, ps) \land \\
\qquad\quad (\exists\, c : Agent \bullet \\
\qquad\qquad \mathsf{achieves}\ (c, g, ps) \land \\
\qquad\qquad (\exists\, p : ps \bullet ((extdes\ c).resourcesofplan\ p) \cap \\
\qquad\qquad\quad (extdes\ b).ownedresources \neq \{\})) \land \\
\qquad \mathsf{sdep}\ (a, b, g, ps) \Leftrightarrow \\
\qquad\quad \mathsf{adep}\ (a, b, g, ps) \lor \ \ \mathsf{rdep}\ (a, b, g, ps) \\
\hline
\end{array}
$$

This reformulation also highlights some difficulties. It makes little sense to say that I *a-depend* on an agent for some goal if the actions that achieve that goal are in my capabilities. Similarly, it makes little sense to say that I *r-depend* on some agent for some resource if that resource is also owned by myself. (It is not made clear in the paper but it is possible that the work is assuming that no two agents can share an action or a resource even though this would be severely limiting.)

A more intuitive definition might be

$$
\begin{array}{l}
\mathsf{adep}\ (a, b, g, ps) \Leftrightarrow \\
\quad (\exists\, c : Agent \bullet \\
\qquad \mathsf{achieves}\ (c, g, ps) \land \\
\qquad (\exists\, a : (\bigcup \{p : ps \bullet \mathrm{ran}\, p\} \bullet \\
\qquad\quad a \in (extdes\ b).capabilities \land \\
\qquad\quad a \notin (extdes\ a).capabilities)))
\end{array}
$$

However, even when an agent is capable of some action of which I am not capable, and which I require for some plan, it still makes little sense to say there is a dependency. It is more appropriate to say that there is a possibility of that agent being able to help in achieving a goal. There is no doubt that such reasoning will be useful in certain situations.

A better notion of actual *dependency* with respect to a goal, would be if *every* plan in the set of plans required some agent's assistance. Thus there would be a real dependency on this agent in order

to achieve the goal. (Note that this is a dependency based on the goals an agent must achieve. It is not a solely action-based notion of dependency.)

$$
\begin{array}{l}
\mathsf{adep}\ (a, b, g, ps) \Leftrightarrow \\
\quad (\exists\, c : Agent \bullet \\
\qquad \mathsf{achieves}\ (c, g, ps) \land \\
\qquad (\forall\, p : ps \bullet \exists\, a : \mathrm{ran}\, p \bullet \\
\qquad\quad a \in (extdes\ b).capabilities \land \\
\qquad\quad a \notin (extdes\ a).capabilities)))
\end{array}
$$

These relations provide an agent with the structures that can be used to reason about others with a view to choosing an appropriate course of action in the context of its dependencies on others' goals, plans, resources, and so on. We believe that social dependence networks constitute an important theoretical basis in studying the nature of the dependencies agents may have on each other, and in this section we have shown how we have been able to isolate inconsistencies and ambiguities in the theory by reformulating it within our framework.

6 Discussion

6.1 Related Work

There is a fair amount of related work in relation to the effort to provide specifications of multi-agent systems, and several approaches to conceptual-level specification have been recently proposed. Unlike the general-purpose formal specification language approach adopted in this paper, DESIRE [5], for example, offers a compositional development method with well-structured compositional designs that is claimed can be specified at a higher level of conceptualisation and implemented automatically using automated prototype generators. Essentially, DESIRE offers an executable specification framework for knowledge-based systems, that can be readily applied to agent-based systems.

In [43], DESIRE is compared with the Concurrent METATEM programming language, another related effort. In Concurrent METATEM, an agent is programmed by giving it an *executable specification* of its behaviour, where such a specification is expressed as a set of temporal logic formulae of the form $past \Rightarrow future$. Execution of these rules proceeds by matching the past time antecedents of temporal logic rules against future time consequents; any rules that fire then become *commitments*, which the agent must subsequently attempt to satisfy. Concurrent METATEM can thus be used to encode a dMARS-like interpreter as a set of Concurrent METATEM rules. The same paper also provides an encoding of an abstract BDI interpreter using DESIRE though here it is less easy to represent the core behaviour of a small but powerful agent in a concise manner.

Other development methods for the specification of multi-agent systems that commit to a specific agent architecture have also been proposed, such as [34] based on the BDI agent architecture, together with object-oriented design methods. A more detailed comparison and evaluation of agent modelling methods, including DESIRE, Z/SMART and others is available in [48, 47].

6.1.1 Conclusions

The lack of a common understanding and structure within which to pursue research in multi-agent systems is set to hamper the further development of the field if efforts are not made to address it. This paper has described one such effort, which aims to provide a framework that will allow the development of diverse models, theories and systems all related within a single unifying whole. The

requirements that we have set out for formal frameworks in general are satisfied by the work reported here, and the adequacy of our formal framework is demonstrated by elaborating and refining it so that we can present models, both existing, well-known implemented systems, and new, theoretical models. In addition, we can construct abstract general models of agents and define the relationships between them so that these concepts may be applied to models at higher levels of detail.

The incremental development of our work has been facilitated in Z by using schema inclusion. This can help ensure that at each new abstraction level, only the necessary details required to define an agent system at that level are introduced and considered. In addition, new information can be formally related to existing information from previous levels. SMART provides a whole range of levels of abstraction that are formally related, enabling the most suitable abstraction level to be more readily selected for the task at hand.

This paper draws together many different aspects of our work, spanning a range of levels of detail and abstraction, and covering many different notions and systems. If the field of multi-agent systems is to progress in a rigorous, disciplined and, perhaps most importantly, accessible way, then efforts such as this, which seek to provide a common unifying foundation for a diverse body of work, are critical.

References

[1] R. Aylett and M. Luck. Applying artificial intelligence to virtual reality: Intelligent virtual environments. *Applied Artificial Intelligence*, 14(1), to appear, 2000.

[2] J. P. Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press, 1996.

[3] J. P. Bowen, S. Fett, and M. G. Hinchey, editors. *ZUM'98: The Z Formal Specification Notation, 11th International Conference of Z Users, Lecture Notes in Computer Science*, volume 1493. Springer-Verlag, 1998.

[4] J. P. Bowen, M. G. Hinchey, and D. Till, editors. *ZUM'97: The Z Formal Specification Notation, 10th International Conference of Z Users, Lecture Notes in Computer Science*, volume 1212. Springer-Verlag, 1997.

[5] F. Brazier, B. Dunin Keplicz, N. Jennings, and J. Treur. Formal specification of multi-agent systems: A real-world case. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 25–32, Menlo Park, 1995. AAAI Press / MIT Press.

[6] J. A. Campbell and M. d'Inverno. Knowledge interchange protocols. In Y. Demazeau and J.-P. Müller, editors, *Decentralized AI: Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 63–80. Elsevier, 1990.

[7] C. Castelfranchi. Social power. In Y. Demazeau and J.-P. Müller, editors, *Decentralized AI — Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-89)*, pages 49–62. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.

[8] C. Castelfranchi, M. Miceli, and A. Cesta. Dependence relations among autonomous agents. In E. Werner and Y. Demazeau, editors, *Decentralized AI 3 — Proceedings of the Third European*

Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-91), pages 215–231. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1992.

[9] B. Chaib-draa. Industrial applications of distributed ai. *Communications of the AMC*, 38(11):49–53, 1995.

[10] B. Chellas. *Modal Logic: An Introduction.* Cambridge University Press: Cambridge, England, 1980.

[11] B. Crabtree. What chance software agents? *Knowledge Engineering Review*, 13(2):131–136, 1998.

[12] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20(1):63–109, 1983.

[13] M. d'Inverno. *Agents, Agency and Autonomy: A Formal Computational Model*. PhD thesis, University College London, University of London, 1998.

[14] M. d'Inverno, M. Fisher, A. Lomuscio, M. Luck, M. de Rijke, M. Ryan, and M. Wooldridge. Formalisms for multi-agent systems. *Knowledge Engineering Review*, 12(3):315–321, 1997.

[15] M. d'Inverno, D. Kinny, and M. Luck. Interaction protocols in agentis. In *ICMAS'98, Third International Conference on Multi-Agent Systems*, pages 112–119, Paris, France, 1998. IEEE Computer Society.

[16] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages*, volume 1365, pages 155–176. Springer-Verlag, 1998.

[17] M. d'Inverno and M. Luck. A formal view of social dependence networks. In C. Zhang and D. Lukose, editors, *Distributed Artificial Intelligence Architecture and Modelling: Proceedings of the First Australian Workshop on Distributed Artificial Intelligence, Lecture Notes in Artificial Intelligence*, volume 1087, pages 115–129. Springer Verlag, 1996.

[18] M. d'Inverno and M. Luck. Engineering agentspeak(l): A formal computational model. *Logic and Computation*, 8(3):233–260, 1998.

[19] M. d'Inverno, M. Priestley, and M. Luck. A formal framework for hypertext systems. *IEE Proceedings - Software Engineering Journal*, 144(3):175–184, June, 1997.

[20] E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'not never' revisited: on branching time versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[21] O. Etzioni, H. M. Levy, R. B. Segal, and C. A. Thekkath. The softbot approach to os interfaces. *IEEE Software*, 12(4), 1995.

[22] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, 1994.

[23] M. R. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, 1987.

[24] R. Goodwin. A formal specification of agent properties. *Journal of Logic and Computation*, 5(6), 1995.

[25] S. Grand and D. Cliff. Creatures: Entertainment software agents with artificial life. *Autonomous Agents and Multi-Agent Systems*, 1(1):39–57, 1998.

[26] R. H. Guttman, A. G. Moukas, and P. Maes. Agent-mediated electronic commerce: a survey. *Knowledge Engineering Review*, 13(2):147–159, 1998.

[27] I. J. Hayes(Editor). *Specification Case Studies.* Prentice Hall, Hemel Hempstead, second edition, 1993.

[28] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[29] N. R. Jennings, P. Faratin, M. J. Johnson, P. O'Brien, and M. E. Wiegand. Agent-based business process management. *International Journal of Cooperative Information Systems*, 5(2 & 3):105–130, 1996.

[30] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[31] N. R. Jennings and T. Wittig. ARCHON: Theory and practice. In *Distributed Artificial Intelligence: Theory and Praxis*, pages 179–195. ECSC, EEC, EAEC, 1992.

[32] W. L. Johnson and B. Hayes-Roth, editors. *Proceedings of the First International Conference on Autonomous Agents.* ACM Press, 1997.

[33] C. B. Jones. *Systematic Software Development using VDM (second edition).* Prentice Hall, 1990.

[34] D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of BDI agents. In Y. Demazeau and J.-P. Müller, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, LNAI 1038*, pages 56–71. Springer-Verlag, 1996.

[35] D. Kuokka and L. Harada. Matchmaking for information agents. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 672–679, Montréal, Québec, Canada, August 1995.

[36] Kevin Lano. *The B Language and Method: A guide to Practical Formal Development.* Springer Verlag, 1996.

[37] Y. Lashkari, M. Metral, and P. Maes. Collaborative interface agents. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 444–449, 1994.

[38] M. Luck. From definition to deployment: What next for agent-based systems? *The Knowledge Engineering Review*, pages 119–124, 1999.

[39] M. Luck and M. d'Inverno. A formal framework for agency and autonomy. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 254–260. AAAI Press / MIT Press, 1995.

[40] M. Luck and M. d'Inverno. A conceptual framework for agent definition and development. *The Computer Journal*, To Appear (2001).

[41] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[42] B. G. Milnes. A specification of the Soar architecture in Z. Technical Report CMU-CS-92-169, School of Computer Science, Carnegie Mellon University, 1992.

[43] M. Mulder, J. Treur, and M. Fisher. Agent modelling in concurrent METATEM and DESIRE. In *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages, LNAI 1365*, pages 193–207. Springer, 1998.

[44] H. Van Dyke Parunak. Applications of distributed artificial intelligence in industry. In G. M. P. O'Hare and N. R. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, pages 139–164. Wiley, 1996.

[45] H. Van Dyke Parunak. What can agents do in industry, and why? an overview of industrially-oriented r&d at cec. In M. Klusch and G. Weiss, editors, *Cooperative Information Agents II, Lecture Notes in Artificial Intelligence 1435*, pages 1–18. Springer, 1998.

[46] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038)*, pages 42–55. Springer-Verlag: Heidelberg, Germany, 1996.

[47] O. Shehory and A. Sturm. Evaluation of agent-based system modeling techniques. Technical Report TR-ISE/IE-003-2000, Faculty of Industrial Engineering and Management Technion – Israel Institute of Technology, 2000.

[48] O. Shehory and A. Sturm. Evaluation of modeling techniques for agent-based systems. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 624–631. ACM Press, 2001.

[49] J. S. Sichman, Y. Demazeau, R. Conte, and C. Castelfranchi. A social reasoning mechanism based on dependence networks. In *ECAI 94. 11th European Conference on Artificial Intelligence*, pages 188–192. John Wiley and Sons, 1994.

[50] J. S. Sichman and Yves Demazeau. Exploiting social reasoning to deal with agency level inconsistency. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 352–359, Menlo Park, 1995. AAAI Press / MIT Press.

[51] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, 29(12):1104–1113, 1980.

[52] R. G. Smith and R. Davis. Frameworks for cooperation in distributed problem solving. *IEEE Transactions on Systems, Man and Cybernetics*, 11(1):61–70, 1981.

[53] M. Spivey. *The Z Notation (second edition)*. Prentice Hall International: Hemel Hempstead, England, 1992.

[54] C. Toomey and W. Mark. Satellite image dissemination via software agents. *IEEE Expert*, 10(5):44–51, 1995.

[55] M. Weber. Combining Statecharts and Z for the design of safety-critical control systems. In M.-C. Gaudel and J. C. P. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 307–326. Formal Methods Europe, Springer-Verlag, 1996.

[56] W. Wen and F. Mizoguchi. Analysis and verification of multi-agent interaction protocols. In *Proceedings of IEEE APSEC'99*, pages 252–259, Takamatsu, Japan, 1999.

[57] D. Wong, N. Paciorek, and D. Moore. Java-based mobile agents. *Communications of the ACM*, 42(3):92–102, 1999.

[58] M. Wooldridge and N. Jennings. The cooperative problem solving aprocess. *Journal of Logic & Computation*, 9, 1999.

[59] M. J. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.